



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola d'Enginyeria de Telecomunicació
i Aeroespacial de Castelldefels

TRABAJO DE FIN DE GRADO

TÍTULO DEL TFG: ColoTe (Contenedores de IoT Empotrados)

TITULACIÓN: Grau en Enginyeria Telemàtica

AUTOR: Alfredo Varela Romero

DIRECTOR: Juan López Rubio

SUPERVISOR: Gerard Solé Castellví

FECHA: 21 de octubre de 2016

Título: ColoTe (Contenedores de IoT Empotrados)

Autor: Alfredo Varela Romero

Director: Juan López Rubio

Supervisor: Gerard Solé Castellví

Fecha: 21 de octubre de 2016

Resumen

En los últimos años, las investigaciones sobre los temas de IoT han avanzado notablemente tanto en dispositivos como en tecnologías. La demanda de este tipo de productos hace que muchas empresas cambien su orientación de mercado y se centren en el mundo de los sensores, incluso invierten en hacer una instalación de este tipo para mejorar su productividad.

Las investigaciones no solo se centran en crear nuevos protocolos o nuevas tecnologías, sino también en reutilizar las que ya se conocen, aplicando pequeñas mejoras o sin cambios, ya que son completamente compatibles.

Por las razones expuestas surge este proyecto, en el que se realizará un nuevo producto que permitirá a la empresa AlterAid disponer de toda su infraestructura de manera remota. De este modo podrá desplegar sus aplicaciones en un lugar en el que el acceso o la conexión a internet sean nulos y/o hayan ocurrido desastres naturales, etc. Paralelamente, la investigación sobre nuevas tecnologías para desplegar aplicaciones, utilizadas sobretodo en el ámbito de servidores, serán llevadas al mundo de IoT para solucionar problemas como el alto consumo recursos o los ocasionados al hacer una actualización del sistema o al hacer nuevos despliegues y, a su vez, reducir los tiempos de éstos.

En el proyecto se utilizará una Raspberry Pi para simular un nodo central o sumidero de datos en la que, mediante Docker, se desplegará toda la infraestructura del core de la empresa AlterAid, Aaaida. Se utilizarán sensores que, mediante bluetooth, se conectarán a la Raspberry Pi para establecer la red, que permitirá el envío de los datos y su correcto almacenamiento para ser gestionados por la aplicación.

Title : ColoTe (Containers of IoT Embedded)

Author: Alfredo Varela Romero

Advisor: Juan López Rubio

Supervisor: Gerard Solé Castellví

Date: October 21, 2016

Overview

In recent years, the progress on IoT-related issues has seen a very high increase in the effort on research, both for devices and technologies. The high demand for these products is forcing many companies to change their market orientation and focus on the world of sensors, or even to invest in facilities related to this topic in order to improve productivity.

Research is not only focused on creating new protocols or new technologies, but also on reusing some which are already known, by applying slight improvements or just keeping them unchanged, as they are fully compatible.

For the reasons presented above, this project is born, in which a new product for a company will be developed, in such a way that the company will have all its infrastructure accessible remotely. Therefore, it will be possible to deploy their applications in a place where, for example, Internet access or connection is null, natural disasters have occurred, etc. In parallel, the research of a new technology to deploy applications will be studied too, being it specifically used in the field of servers, but also taken to the world of IoT to solve problems such as the high resources consumption when doing a system update, making new deployments and reducing their times.

In the project, a Raspberry Pi is used to simulate a central node or sink of data which, by using Docker, can deploy all the core infrastructure of the AlterAid enterprise (Aaaida). Some sensors will also be used, and by means of Bluetooth they connect to the Raspberry Pi, establishing a network which will allow the sending of data and its proper storage to be managed by the corresponding applications.

ÍNDICE GENERAL

Introducción	1
CAPÍTULO 1. Visión General del proyecto	3
1.1. Docker	3
1.2. Aaaida	3
1.3. Motivación	4
1.4. Objetivos	4
1.5. Organización del proyecto	5
CAPÍTULO 2. Virtualización	7
2.1. ¿Qué es la virtualización?	7
2.1.1. Tipos de virtualización	7
2.1.2. Ventajas de la virtualización	8
2.2. ¿Qué es Docker?	8
2.3. Máquinas Virtuales vs Docker	9
2.4. ¿Por qué Docker?	10
2.5. ¿Cómo funciona Docker?	10
2.5.1. Arquitectura	10
2.5.2. Creación de Imágenes	12
2.5.3. Docker Compose	15
2.6. Conclusiones	17
CAPÍTULO 3. Raspberry Pi	19
3.1. ¿Por qué Raspberry Pi?	19
3.2. Virtualización de la Raspberry Pi	20
3.3. Raspberry Pi y Docker	20
3.3.1. Instalación de la imagen en la Raspberry Pi	20
3.3.2. Creación de las imágenes de Docker	21

3.3.3. Despliegue de la aplicación	23
3.3.4. Conclusiones	26
CAPÍTULO 4. Aaaida	27
4.1. Arquitectura	27
4.1.1. Stack MEAN	27
4.1.2. API REST	28
4.1.3. Consola de Aaaida	28
CAPÍTULO 5. Implementación	31
5.1. Comunicación con el sensor	31
5.1.1. Protocolo	32
5.2. Integración con Aaaida	35
5.2.1. Creación de un plugin	35
5.2.2. Creación de la página	37
5.2.3. Recursos utilizados	39
5.3. Despliegue en la Raspberry Pi	39
5.4. Conclusiones y Resultados	43
Conclusiones	45
5.5. Conclusiones del proyecto	45
5.6. Conclusiones personales	45
5.7. Implementaciones futuras	46
5.8. Impacto ambiental	46
Glosario	47
Bibliografía	49
APÉNDICE A. Emular la Raspberry Pi en Qemu	53
APÉNDICE B. Ficheros utilizados	55
B.1. Fichero entrypt	55

B.2. Script build rpi image 55

B.3. Dockerfile modificado 56

B.4. Docker-compose.yml modificado 57

ÍNDICE DE FIGURAS

1.1	Logotipo de Docker	3
1.2	Logotipo de Aaaida	4
2.1	Comparativa de máquina virtual y Docker	9
2.2	Infraestructura de Docker	11
2.3	Comandos de Docker (I)	12
2.4	Comandos de Docker (II)	13
2.5	Build del DockerFile	14
2.6	Docker images	14
2.7	Docker run docker-whale	15
3.1	Raspberry Pi 3	19
3.2	Docker-compose up	23
3.3	Listado de contenedores y volúmenes	24
3.4	Acceso a Aaaida	24
3.5	Docker inspect	25
3.6	Interfaz de Aaaida	25
4.1	Arquitectura de Aaaida	27
4.2	Stack MEAN	27
5.1	Sesor Zephyr BioHarness 3	31
5.2	Arquitectura de los plugins	35
5.3	Arquitectura del directorio pages	38
5.4	Sección en la consola de Aaaida	38
5.5	Visualización de los datos en Aaaida	39
5.6	Build del script	41
5.7	Creación de contenedores y ejecución	41
5.8	Tomar la medida	42
5.9	Conexión con el sensor	42
5.10	Guardar en la base de datos	42
5.11	Visualización del resultado	43

ÍNDICE DE CUADROS

5.1 Tabla de rutas 39

INTRODUCCIÓN

En los últimos años, las investigaciones sobre los temas de IoT han avanzado notablemente tanto en dispositivos como en tecnologías. Con el objetivo de superar las barreras impuestas por el software y el hardware disponibles, se han conseguido protocolos muy simples y sensores extremadamente pequeños y baratos. La demanda de este tipo de productos hace que muchas empresas cambien su orientación de mercado y se centren en el mundo de los sensores, incluso invierten en hacer una instalación de este tipo para mejorar su productividad.

Por eso se decidió realizar este proyecto, en el que se llevará a cabo el despliegue de la plataforma Aaaida en una Raspberry Pi. Ésta nos puede permitir el uso de dicha plataforma y sus aplicaciones en lugares en los cuales la conexión a internet es nula, en lugares que hayan sufrido un desastre natural, etc. Aaaida es una plataforma centrada en eHealth, que nos permite la monitorización del estado de un paciente, cosa que afirma la necesidad de poder utilizarla en los casos de emergencia nombrados anteriormente. Gracias a las conexiones de la Raspberry Pi, se puede realizar una red con sensores que mediante Bluetooth se puedan comunicar y puedan almacenar los datos.

Por otra parte, el despliegue de la aplicación se llevará a cabo usando Docker, un software que permite añadir en contenedores todo aquello que nuestras aplicaciones necesiten. Éste será muy útil, ya que nos permite instalar o actualizar contenedores por separado y no tener que hacerlo para todo el sistema. Docker es utilizado básicamente en el mundo de servidores y no en IoT. Las ventajas que nos proporcionará a la hora del despliegue son muchas pero hay que tener en cuenta su complejidad en sistemas con capacidades limitadas.

Nuestra intención, es poder desplegar toda la infraestructura de la empresa utilizando un sistema nuevo para IoT y poder comprobar su viabilidad y sus beneficios. Se llegará a ofrecer un producto que podría ser de gran utilidad en caso de desastres, como sería un sistema de monitorización de pacientes mediante sensores.

La utilización de la Raspberry Pi, que es un dispositivo muy utilizado en IoT por sus grandes prestaciones a bajo precio, puede servir como aproximación y prueba piloto para probar la viabilidad de la tecnología nombrada anteriormente. Docker podría abrirse un hueco en el mundo de IoT, aunque originalmente está orientado a arquitecturas de servidor con recursos ilimitados, pero puede ser utilizado perfectamente con los recursos que nos ofrece una Raspberry Pi.

El sensor utilizado para realizar este proyecto será una banda HRM (Heart Rate Monitor), más concretamente el Zephyr BioHarness 3, un dispositivo con conexión bluetooth con el que se podrá realizar la red de comunicación y generar los datos para gestionarlos en la plataforma de Aaaida.

CAPÍTULO 1. VISIÓN GENERAL DEL PROYECTO

El propósito de este proyecto es desplegar toda la infraestructura de la empresa AlterAid en un dispositivo móvil, en nuestro caso una Raspberry Pi, con la finalidad de poder conseguir desplegar las aplicaciones en lugares remotos, sin conexión a internet y/o que han sufrido un desastre natural.

Como segundo objetivo del proyecto queremos implementar un pequeño despliegue de sensores haciendo que nuestra Raspberry Pi sea el nodo central o sumidero de datos. Para poder llevar a cabo todo esto es necesario contar con la utilización de los contenedores usando la tecnología Docker. Puesto que éstos nos facilitarán el trabajo, su utilización fuera del mundo de servidores es motivo de investigación.

1.1. Docker

La idea detrás de Docker es crear contenedores ligeros y portables para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que la máquina tenga por debajo, facilitando así también los despliegues.



Figura 1.1: Logotipo de Docker

1.2. Aaaida

Es una plataforma desarrollada por la empresa AlterAid que nos permite crear un usuario y con este usuario poder tener varios vínculos. Un Vínculo es un ente que representa aquello por el que el Usuario se ocupa y preocupa. Ejemplos de Vínculos de Usuario pueden ser familiares, pacientes, amigos, etc. Ésta es una plataforma web que recoge todos los datos de las aplicaciones de la empresa y los almacena. Por eso la importancia de poder desplegar toda la infraestructura en un dispositivo que sea fácil de transportar y poder llegar a cualquier lugar.



Figura 1.2: Logotipo de Aaaida

1.3. Motivación

Conseguir el despliegue de toda la infraestructura de una empresa en un dispositivo de reducidas prestaciones y tamaño como sería una Raspberry Pi permitiría la apertura de nuevos campos de investigación y de mercado puesto que se podrían desarrollar otro tipo de aplicaciones para casos de emergencia o lugares con pocos medios. Utilizar una tecnología de servidores como es Docker conlleva un gran avance en el mundo de Internet of Things (de ahora en adelante IoT). El presente proyecto representa una prueba de concepto para versiones futuras en las cuales se podrían utilizar estas tecnologías para facilitar el despliegue tanto de las aplicaciones como de sus actualizaciones ya que al utilizar Docker solo se debería actualizar el contenedor independiente y no todo el sistema.

1.4. Objetivos

Los objetivos fijados para el desarrollo de este proyecto son los siguientes:

- Analizar y escoger las tecnologías a utilizar para el desarrollo del proyecto
- La instalación de Docker en una Raspberry Pi
- El despliegue de la infraestructura de la empresa AlterAid
- La creación de pequeños nodos
- El despliegue de la red
- Contemplar la viabilidad de creación de redes smart con estas tecnologías.

1.5. Organización del proyecto

En primer lugar, para poder cumplir todos los puntos de los objetivos necesitaremos estudiar un poco más a fondo todas las tecnologías que se utilizarán a lo largo de todo el proyecto. Serán explicadas en sus respectivos capítulos.

Como se ha explicado previamente se utilizará una Raspberry Pi, donde se desplegará la aplicación mediante Docker. Para poder llevarlo a cabo se tendrán que estudiar las diferentes posibilidades como, por ejemplo, si es posible ejecutar Docker en un sistema ARM o si se podrá virtualizar la Raspberry Pi para poder hacer las pruebas de una manera más cómoda. Todo esto se podrá ver en el capítulo 2 y capítulo 3.

Una vez desplegada y ejecutada la aplicación es necesario arrancar Aaaida. En el capítulo 4 se podrá ver una pequeña explicación del funcionamiento de la plataforma y sus funcionalidades.

Para terminar la parte técnica, vendría el paso de crear la red de sensores y comunicarnos con la Raspberry Pi. Una vez se haya llevado a cabo esta conexión, se realizará el envío de los datos y su correcto almacenamiento. En el capítulo 5 se comentará como hacerlo para poder ser visualizados en la plataforma Aaaida.

Por último, con la plataforma en la Raspberry y la red de sensores funcionando obtendremos el último capítulo donde se podrán ver las conclusiones y resultados sobre la viabilidad de este proyecto.

CAPÍTULO 2. VIRTUALIZACIÓN

2.1. ¿Qué es la virtualización?

La virtualización es la creación de una versión virtual (no física) de algo. Está basada en software, se puede aplicar a sistemas operativos, almacenamiento, servidores, aplicaciones, redes, etc. y es una manera de reducir gastos y aumentar eficiencia y agilidad en las empresas.

2.1.1. Tipos de virtualización

Estos son los 4 tipos de virtualización más habituales.

2.1.1.1. *Virtualización de servidores*

La virtualización de servidores ayuda a evitar ineficiencias ya que permite ejecutar varios sistemas operativos en una máquina física con máquinas virtuales con acceso a los recursos de todos. También permite generar un clúster de servidores en un único recurso para así mejorar mucho más la eficiencia y la reducción de costes. También permite el aumento de rendimiento de las aplicaciones y la disponibilidad al aumentar la velocidad en la carga de trabajo.

2.1.1.2. *Virtualización de escritorios*

La implementación de escritorios virtualizados permite ofrecer a las sucursales o empleados externos de forma rápida y sencilla un entorno de trabajo y una reducción de la inversión a la hora de gestionar cambios en éstos.

2.1.1.3. *Virtualización de red*

Se trata de reproducir una red física completa mediante software para poder ejecutar los mismos servicios que una red convencional y sus dispositivos. Cuentan con las mismas características y garantías que las redes físicas con las ventajas que nos ofrece la virtualización además de la liberación del hardware.

2.1.1.4. *Almacenamiento definido por software*

La virtualización del almacenamiento permite prescindir de los discos de los servidores. Los combina en depósitos de almacenamiento de alto rendimiento y los distribuye como software. Este nuevo modelo permite aumentar la eficiencia en el guardado de datos.

2.1.2. Ventajas de la virtualización

Como se ha podido apreciar en los tipos de virtualización presentados anteriormente, ésta conlleva una mejora considerable tanto en el rendimiento, agilidad, flexibilidad, escalabilidad, etc. como en una reducción considerable de los costes económicos y de tiempo y una simplificación en la gestión de la infraestructura.

- Reduce los costes de capital y los gastos operativos.
- Minimiza o elimina los tiempos de inactividad.
- Aumenta la productividad, la eficiencia, la agilidad y la capacidad de respuesta.
- Implementa aplicaciones y recursos con más rapidez.
- Garantiza la continuidad del negocio y la recuperación ante desastres.
- Simplifica la gestión del centro de datos.

2.2. ¿Qué es Docker?

Docker es un proyecto Open Source basado en contenedores de Linux. Es básicamente un motor de contenedores que usa características del Kernel de Linux.

La idea detrás de Docker es crear contenedores ligeros y portables para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que la máquina tenga por debajo, facilitando así también los despliegues.

De una manera más sencilla, Docker proporciona la opción de introducir en pequeños contenedores todo aquello que nuestra aplicación necesite. Permite desplegarla en cualquier máquina que tenga Docker instalado, sin preocuparnos de nada más.

Se podría decir que Docker son pequeñas “máquinas virtuales” pero muchos más ligeras ya que utilizan el sistema operativo de donde se ejecuta y el contenido relevante para ejecutar la aplicación está dentro de los contenedores.

Docker es:

- Open-Source para la gestión de “virtualización de contenedores”
- Aísla múltiples sistemas de archivos en el interior del mismo host
 - Las instancias se llaman Contenedores
 - Te dan la ilusión de estar dentro de una máquina virtual
- Pensado para entornos de ejecución o “sandboxes”
- No hay necesidad de un hipervisor (rápido de ejecutar)
- Requiere x64 y Linux kernel 3.8+

Docker no es:

- Un lenguaje de programación
- Un sistema operativo
- Una máquina virtual
- Una imagen en el concepto tradicional de la máquina virtual basada en hipervisor

2.3. Máquinas Virtuales vs Docker

Las máquinas virtuales incluyen toda la aplicación, los binarios y librerías necesarias y todo un sistema operativo. Esto implica que ocupen mucho espacio, que el tiempo de ejecución sea lento y la necesidad de un Hipervisor para su utilización.

Por lo contrario, los Docker Container incluyen la aplicación y todas sus dependencias pero comparten el núcleo con otros contenedores, funcionando como procesos aislados en el sistema de ficheros del sistema operativo. Los Docker container no están vinculados a ninguna infraestructura específica: se ejecutan en cualquier ordenador, en cualquier infraestructura y en cualquier cloud.

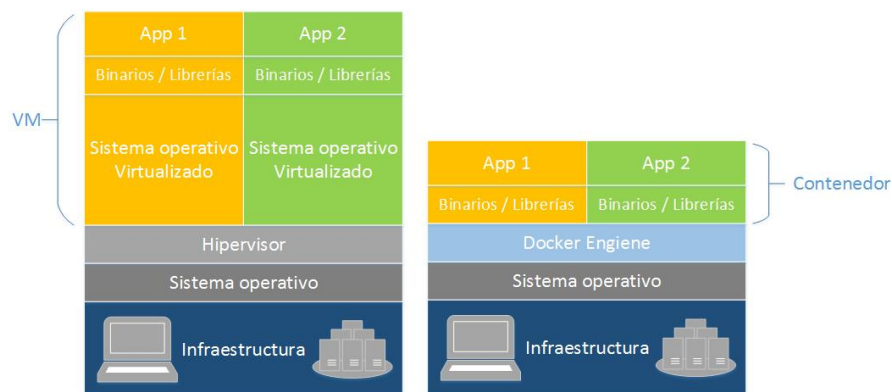


Figura 2.1: Comparativa de máquina virtual y Docker

En la figura 2.1 se aprecian las diferencias comentadas anteriormente. La primera columna corresponde a la virtualización de 2 aplicaciones mediante la capa intermedia del Hipervisor, el cual nos permite ejecutar los sistemas operativos virtualizados. Estos, a su vez, permiten ejecutar todos los binarios y librerías que requiere cada aplicación y, finalmente, la aplicación. La segunda columna corresponde a la contenerización de 2 aplicaciones mediante Docker Engine. Podemos observar que no hace falta un hipervisor ya que utilizan el sistema operativo de la infraestructura de donde son ejecutadas, pero sí son necesarios los binarios y las librerías. A simple vista se puede apreciar que, si eliminamos el sistema operativo, la infraestructura completa se hace más liviana y rápida de ejecutar.

2.4. ¿Por qué Docker?

Según lo listado en los apartados anteriores, se decidió utilizar Docker para realizar el despliegue de las aplicaciones por:

Cumplir con la condición de tecnología que pueda ser almacenada en dispositivos con una memoria reducida.

Su rapidez a la hora de levantar el servicio. En el mundo de IoT el tiempo es un bien preciado y los sistemas se pueden apagar y encender constantemente para evitar gastos de energía innecesarios.

La facilidad para poder desplegar los servicios. Con Docker desplegar servicios es muy sencillo, solo requiere tenerlo instalado y ejecutar el contenedor para que el servicio esté activo.

La sencillez a la hora de mantener el sistema. Si hay que hacer actualizaciones o controles de una pequeña parte del servicio solo habría que cambiar o actualizar ese contenedor y no todo el sistema. Esto es un gran ahorro en recursos y tiempo.

Por todos estos motivos se piensa que Docker puede ser una buena tecnología para desplegar sistemas de IoT. También hay que tener en cuenta que, aunque la Raspberry Pi no es un aparato con unas capacidades limitadas como podría ser un sensor utilizado normalmente, sí que cumple con todas las funciones listadas anteriormente y, por lo tanto, puede servir como preámbulo para la utilización en el resto de despliegues.

2.5. ¿Cómo funciona Docker?

En este punto se explicarán brevemente aspectos de la arquitectura, funcionamiento y puesta en marcha de Docker.

2.5.1. Arquitectura

Docker usa una arquitectura cliente-servidor. El cliente de Docker se comunica con el Daemon de Docker para crear, ejecutar y distribuir los contenedores. Tanto el cliente como el Daemon pueden estar en el mismo sistema o pueden conectarse remotamente. Como Docker usa el kernel de Linux para su ejecución, si el sistema operativo del sistema no es éste, se deberá usar una pequeña capa extra en la arquitectura de tipo VM (boot Docker) para poder correr Docker en la máquina.

2.5.1.1. Cliente de Docker

Es la principal interfaz de usuario para Docker. Acepta los comandos del usuario y se comunica con el Daemon de Docker.

2.5.1.2. Imágenes de Docker (Docker Images)

Las imágenes de Docker son plantillas de sólo lectura, que nos permitirán crear contenedores basados en su configuración.

2.5.1.3. Registros de Docker (Docker Registries)

Los registros de Docker guardan las imágenes. Éstos son repositorios públicos o privados donde se pueden subir o descargar imágenes. Sería similar a GitHub para imágenes de Docker (Docker Hub).

2.5.1.4. Contenedores de Docker (Docker Containers)

El contenedor de Docker contiene todo lo necesario para ejecutar una aplicación. Cada contenedor se crea de una imagen de Docker y es una plataforma aislada.

En la figura 2.2 podemos apreciar gráficamente cómo sería la arquitectura básica. El cliente podría hacer los comandos básicos de docker:

Docker Build: Hacer un build de un DockerFile y generar una imagen de Docker. En la figura está representado siguiendo las flechas rojas en las cuales podemos ver que el cliente se comunica con el Daemon y éste genera la imagen.

Docker Pull: Permite descargar una imagen de los repositorios de Docker. En la imagen se puede observar siguiendo las flechas verdes e, igual que el comando anterior, se comunica el cliente con el Daemon para que éste proceda a hacer la descarga de la imagen de mongoDB del repositorio.

Docker Run: Ejecuta una imagen para generar un contenedor de ésta. Siguiendo las flechas azules, veremos cómo nuevamente el cliente, al ejecutar esa comanda, se comunica con el Daemon. Este busca la imagen que se quiere ejecutar, en este caso la imagen de Ubuntu y se levanta un contenedor con la configuración de la imagen.

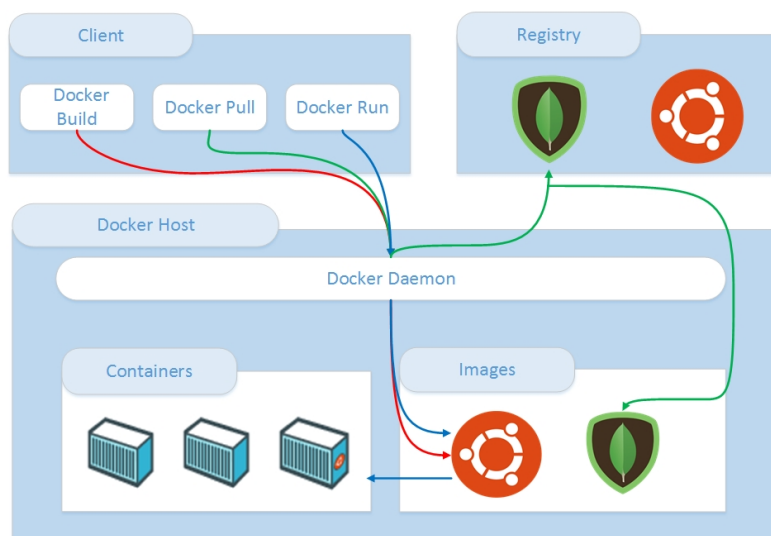


Figura 2.2: Infraestructura de Docker

2.5.2. Creación de Imágenes

Como se comenta en el punto anterior, las imágenes de Docker son las plantillas para poder levantar los contenedores. Por eso la importancia de saber crear imágenes y personalizarlas ya que sólo permiten lectura y los cambios que hagamos en los contenedores no se verán reflejados en éstas.

La manera más sencilla de crear una imagen es descargarla del Docker Hub con el comando explicado anteriormente:

```
docker pull [OPTIONS] NAME[:TAG|@DIGEST]
```

Este comando nos permite descargar una imagen en una versión concreta o tag dependiendo de nuestras necesidades. Por defecto, si no se pone nada, descargará la última.

```
alfredo@alfredo:~$ docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
8ad8b3f87b37: Pull complete
5947be99d359: Pull complete
d5a4577c6007: Pull complete
acb97586a200: Pull complete
d11260d069a3: Pull complete
bf102d35e390: Pull complete
f4964f6a9bfa: Pull complete
8b392ba3e8bf: Pull complete
c023b73abe56: Pull complete
Digest: sha256:8ff7bd4acdb123e3922a7fae7f73efa35fba35af33fad0de946ea31370a23cc4
Status: Downloaded newer image for mongo:latest
alfredo@alfredo:~$
```

(a) Docker pull

```
alfredo@alfredo:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mongo                latest             48b8b08dca4d       2 weeks ago        366.4 MB
alfredo@alfredo:~$
```

(b) Docker images

Figura 2.3: Comandos de Docker (I)

En la figuras 2.3 podemos apreciar como se descarga la última versión de la imagen de MongoDB y nos genera la imagen.

Una vez obtenida la imagen se pasará a levantar el contenedor para poder ejecutar el servicio con otro de los comandos explicados.

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

```
alfredo@alfredo:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS          NAMES
alfredo@alfredo:~$
```

(a) Docker ps

```
alfredo@alfredo:~$ docker run mongo
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten] MongoDB starting : pid=1 port=27017 dbpath=/data/db 64-bit host=d62f7dce38c2
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten] db version v3.2.9
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten] git version: 22ec9e93b40c85fc7cae7d56e7d6a02fd811088c
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten] OpenSSL version: OpenSSL 1.0.1t  3 May 2016
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten] allocator: tcmalloc
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten] modules: none
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten] build environment:
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten]   distmod: debian81
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten]   distarch: x86_64
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten]   target_arch: x86_64
2016-09-19T14:54:09.623+0000 I CONTROL [initandlisten] options: {}
2016-09-19T14:54:09.631+0000 I STORAGE [initandlisten] wiredtiger_open config: create,cache_size=3G,session_max=20000,eviction=(threads_max=4),
config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manager=(close_idle_time=100000),check
points=(wait=60,log_size=2GB),statistics_log=(wait=0),
2016-09-19T14:54:10.421+0000 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/enabled is 'always'.
2016-09-19T14:54:10.421+0000 I CONTROL [initandlisten] We suggest setting it to 'never'
2016-09-19T14:54:10.421+0000 I CONTROL [initandlisten]
2016-09-19T14:54:10.421+0000 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/transparent_hugepage/defrag is 'always'.
2016-09-19T14:54:10.421+0000 I CONTROL [initandlisten] We suggest setting it to 'never'
2016-09-19T14:54:10.421+0000 I CONTROL [initandlisten]
2016-09-19T14:54:10.422+0000 I FTDC [initandlisten] Initializing full-time diagnostic data capture with directory '/data/db/diagnostic.data'
2016-09-19T14:54:10.422+0000 I NETWORK [HostnameCanonicalizationWorker] Starting hostname canonicalization worker
2016-09-19T14:54:10.492+0000 I NETWORK [initandlisten] waiting for connections on port 27017
```

(b) Docker run

```
alfredo@alfredo:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS          NAMES
eee37a9c1602   mongo         "/entrypoint.sh mongo"   4 seconds ago    Up 3 seconds    27017/tcp      kickass_pike
alfredo@alfredo:~$
```

(c) Docker ps

Figura 2.4: Comandos de Docker (II)

En la figuras 2.4 se puede apreciar cómo utilizando el comando `docker ps` que permite ver qué contenedores están levantados, no hay ninguno (a) y cómo al inicializar con el comando `docker run mongo` levanta el servicio (b) y esta vez sí aparece el contenedor (c).

La segunda manera de crear y personalizar imágenes es mediante un `DockerFile`, que es un documento de texto donde se encuentran los comandos que se deben ejecutar para generar nuestra imagen.

El comando `docker build` comunica al Daemon de Docker que debe de leer el `DockerFile` del directorio actual y seguir las instrucciones línea por línea para la creación de nuestra imagen. Este proceso va pintando los resultados por pantalla y generando imágenes intermedias para obtener así una caché que nos permitirá en caso de errores, una vez corregido el `DockerFile`, continuar desde el punto conflictivo.

```
FROM docker/whalesay:latest
CMD echo "Proyecto CoIoTe" | cowsay
```

Aquí tenemos un ejemplo sencillo de `DockerFile` que nos servirá para explicar de una manera rápida como crearlos.

`FROM` indica la imagen base que va a utilizar para seguir futuras instrucciones. Buscará si la imagen se encuentra localmente, en caso de que no, la descargará. En nuestro ejemplo utiliza la última versión de la imagen `docker/whalesay`.

La instrucción `CMD` solo puede aparecer una vez en un `DockerFile`, si colocamos más de uno, solo el último tendrá efecto. El objetivo de esta instrucción es proveer valores por defecto a un contenedor. Estos valores pueden incluir un ejecutable u omitir un ejecutable

que en dado caso se debe especificar un punto de entrada o entrypoint en las instrucciones. En nuestro caso pintaremos un texto que se le pasara a la aplicación Cowsay.

Existen muchas más instrucciones, algunas de éstas serán explicadas más adelante cuando sea necesario su uso.

Una vez ejecutado el comando `docker build -t docker-whale .` donde el `-t` nos permite darle nombre a la imagen y el `.` encontrar el DockerFile para ser compilado, tal y como se ve en la Figura 2.5, descarga del repositorio la imagen ya que no la tenemos en local. Creará una imagen intermedia que nos proporciona la caché en caso de fallo, continuará con la siguiente orden creando una nueva imagen y borrando las anteriores hasta obtener la imagen definitiva.

```
alfredo@alfredo:~/mydockerbuild$ sudo docker build -t docker-whale .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM docker/whalesay:latest
latest: Pulling from docker/whalesay
e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
00bf65475aba: Pull complete
c57b6bcc83e3: Pull complete
8978f6879e2f: Pull complete
8eed3712d2cf: Pull complete
Digest: sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
Status: Downloaded newer image for docker/whalesay:latest
--> 6b362a9f73eb
Step 2 : CMD echo "Proyecto CoIoT" | cowsay
--> Running in 598204c0e3a3
--> ac80256777e9
Removing intermediate container 598204c0e3a3
Successfully built ac80256777e9
```

Figura 2.5: Build del DockerFile

Una vez hecho el build del DockerFile podemos comprobar que nuestra imagen está creada correctamente (Figura 2.6) y pasará a levantar el contenedor. Una vez levantado nos saldrá por pantalla el icono de Docker “diciendo” la frase que le indicamos en el fichero. (Figura 2.7)

```
Successfully built ac80256777e9
alfredo@alfredo:~/mydockerbuild$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker-whale	latest	ac80256777e9	About a minute ago	247 MB
mongo	latest	48b8b08dca4d	2 weeks ago	366.4 MB
docker/whalesay	latest	6b362a9f73eb	16 months ago	247 MB

```
alfredo@alfredo:~/mydockerbuild$
```

Figura 2.6: Docker images



Figura 2.7: Docker run docker-whale

2.5.3. Docker Compose

Docker Compose es un orquestador que nos permite ejecutar aplicaciones que utilicen varios contenedores a la vez. Se creará un archivo `docker-compose.yml` donde se configurarán todos los servicios necesarios para nuestra aplicación. Una vez ejecutado este archivo nos generará todas las imágenes y con éstas los contenedores especificados a la vez que arrancará la aplicación.

Los comandos que se utilizarán serán similares a los utilizados en la creación de imágenes.

Para ejecutar el servicio y levantar todos los contenedores.

```
docker-compose up
```

Para detener el servicio y detener los contenedores.

```
docker-compose stop
```

Docker-compose también permite ejecutar solo partes del `.yml` para levantar y/o detener solo algunos de los contenedores, pasándoles por parámetro el nombre del contenedor.

```
version: '2'
services:
  mongo:
    container_name: mongo
    restart: always
    image: partlab/ubuntu-arm-mongodb
    volumes:
      - mongo-data:/data/db
    command: /usr/bin/mongod --smallfiles --journal
```

```

aaaaida:
  container_name: aaaaidaArm
  restart: always
  image: alteraid/aaaaida-datastore-arm
  links:
    - mongo:mongo
  environment:
    - NODE_ENV=docker
  ports:
    - "40000:40000"
volumes:\newline
mongo-data:
  driver: local

```

Este es un ejemplo de Docker-compose, exactamente el utilizado en el proyecto para poder levantar la infraestructura de Aaaida en la Raspberry Pi. La importancia en los espacios es imprescindible para que éste funcione ya que da la jerarquía adecuada para que docker-compose lo entienda. Ahora pasaremos a explicar brevemente los argumentos que podemos encontrar en él. En primer lugar tenemos los servicios, en nuestro caso son 2, una base de datos MongoDB y Aaaida.

Dentro de `mongo` tendremos los siguientes:

`container name`: Le da un nombre al contenedor para no tener que referirse a él por la id o el nombre aleatorio que proporciona Docker.

`restart`: Nos permite que en caso de falla o reinicio del sistema este contenedor vuelva a ejecutarse.

`image`: Para poder especificar la imagen que utilizaremos para este servicio.

`volumes`: En caso de Mongo necesita un directorio donde almacenar los datos. Ésto especifica donde se creará el volumen de datos.

`command`: El comando que le pasamos al contenedor al momento de su ejecución. En este caso concreto es muy importante ejecutar este comando ya que activará el `Journaling`, que por defecto viene desactivado.

En el siguiente servicio, el de `aaaaida`, a parte de disponer los mismos argumentos básicos como serían `container name`, `restart` o `image`, encontraremos alguno más como:

`links`: Define el enlace entre contenedores.

`environment`: Para poder pasar variables.

`ports`: Define el mapeo de puertos.

Para finalizar el archivo se encuentra los `volumes` donde se genera el volumen que hemos especificado dentro del servicio de `mongo`.

En puntos posteriores se explicará más detenidamente el funcionamiento y la puesta en marcha del Docker Compose ya que el archivo utilizado para el ejemplo es el utilizado en la Raspberry Pi.

2.6. Conclusiones

Como se puede observar, al virtualizar los recursos disponibles obtenemos un aumento en rendimiento y una reducción de los costes.

Docker nos permite el despliegue de aplicaciones de una manera sencilla y en cualquier plataforma que lo soporte.

La diferencia más significativa entre las máquinas virtuales y Docker sería el uso del Hipervisor y la necesidad de un sistema operativo para las máquinas virtuales, cosa que en Docker no es necesario. Ésto hace que las aplicaciones desplegadas de esta manera tengan una reducción en tamaño y en tiempo de ejecución.

CAPÍTULO 3. RASPBERRY PI

3.1. ¿Por qué Raspberry Pi?

Los motivos por los cuales se ha utilizado una Raspberry Pi en este proyecto son claros y han sido comentados con anterioridad. Es un dispositivo portátil de un tamaño muy reducido que nos facilita mucho el poder llevarlo y poder ofrecer un servicio en cualquier lugar sin dificultades. Tiene unas prestaciones más que aceptables para un dispositivo de ese tamaño como conexiones inalámbricas de Wifi y Bluetooth integrados, cosa imprescindible para las comunicaciones con los sensores. Todo a un precio muy atractivo que lo hace asequible.

Algunas de las especificaciones mas importantes de las Raspberry Pi 3 son las siguientes:

- 1.2GHz 64-bit quad-core ARMv8 CPU
- 802.11n Wireless LAN
- Bluetooth 4.1 y Bluetooth Low Energy (BLE)
- 1GB RAM
- Ethernet port
- Micro SD card slot



Figura 3.1: Raspberry Pi 3

3.2. Virtualización de la Raspberry Pi

Al inicio del proyecto no se disponía de una Raspberry Pi para poder realizar las pruebas y se decidió emularla mediante Qemu. Qemu es una aplicación que nos permite emular, mediante máquinas virtuales, gran parte de arquitecturas existentes. Es realmente sencillo emular una imagen de Raspbian Pi si no fuera por un detalle, como se comenta en el apartado 2.2: ¿Qué es Docker? Docker necesita un sistema x64 o Linux kernel 3.8+ mientras que Raspberry Pi ejecuta un sistema ARM, lo que conlleva que Docker no sea compatible a primera instancia con la Raspberry Pi. La solución para este gran problema es la utilización de Hypriot, una imagen de Raspbian modificada con Docker instalado. La instalación de esta imagen requiere hacer unos retoques ya que el kernel de Qemu no es del todo compatible con el de la imagen de Hypriot. Desafortunadamente, las incompatibilidades del kernel no permitían hacer un uso correcto de modo que se decidió apartar por completo la posibilidad de poder emular una Raspberry Pi con Docker instalado ya que las incompatibilidades del kernel no lo permitían.

En el apéndice se podrán ver los pasos a seguir para la instalación y emulación de la imagen.

3.3. Raspberry Pi y Docker

Después de descartar completamente la opción de emular la imagen en Qemu se decidió probar en la Raspberry Pi de la empresa si la imagen corría correctamente. Y sí, la imagen iba perfectamente y ejecutaba Docker sin ningún problema. Contrariamente, la Raspberry Pi sí que dio problemas porque era el primer modelo, que era antiguo, y no disponía de módulos para conexiones inalámbricas integrados por lo que se decidió, aprovechando el lanzamiento de la nueva Raspberry Pi 3, comprarla ya que ésta sí dispone de conexiones inalámbrica.

3.3.1. Instalación de la imagen en la Raspberry Pi

Como se ha dicho, a la imagen de Raspbian no es posible instalar Docker como lo haríamos en un sistema operativo como linux. Es necesario una imagen la cual ya lo tenga instalado, como la imagen de Hypriot. La instalación se llevó a cabo de esta manera:

```
$ curl -sSL http://downloads.hypriot.com/docker-hypriot_1
.8.2-1_armhf.deb >/tmp/docker hypriot_1.8.2-1_armhf.deb
$ sudo dpkg -i /tmp/docker-hypriot_1.8.2-1_armhf.deb
$ rm -f /tmp/docker-hypriot_1.8.2-1_armhf.deb
$ sudo sh -c 'usermod -aG docker $SUDO_USER'
$ sudo systemctl enable docker.service
```

- Primero descargó la imagen deseada y la montará en un directorio temporal.
- Instalará los paquetes con el comando dpkg.

- Borrará el archivo comprimido.
- Ejecutará el comando donde añade el usuario a un grupo Docker.
- Pondrá en marcha el Daemon de Docker.

3.3.2. Creación de las imágenes de Docker

Una vez tenemos la Raspberry lista, se pasará a la creación de las imágenes Docker para poder desplegar Aaaida. En primer lugar se necesita un contenedor que contenga una base de datos, en este caso MongoDB. Para la creación de la imagen utilizaremos una ya creada y que esté en los repositorios de Docker Hub. Se necesita una imagen que sea compatible con ARM, cosa que no resulta sencillo ya que la gran mayoría de las imágenes no están pensadas para esta arquitectura y los pocos disponibles no estaban operativos. Otra restricción que tenemos era que debe ser una base de datos sin la necesidad de ingresar un usuario y su contraseña, cosa que no es habitual, pero que resulta una obligación en algunas imágenes. Por lo tanto se utilizó la siguiente imagen de Docker Hub:

partlab/ubuntu-arm-mongodb

La otra imagen que se utilizará será la propia de Aaaida, pero igual que, la imagen de MongoDB deberá ser compatible con una arquitectura ARM. Al ser una aplicación propia de la empresa no se encontrará en repositorios público y tendremos que crearla nosotros mediante un Dockerfile. En la siguiente figura se puede ver el Dockerfile que se utilizará para poder crear el contenedor con Aaaida.

```
FROM ioft/armhf-debian
RUN apt-get update; apt-get -y install curl
RUN set -ex \
    && for key in \
        9554F04D7259F04124DE6B476D5A82AC7E37093B \
        94AE36675C464D64BAFA68DD7434390BDBE9B9C5 \
        0034A06D9D9B0064CE8ADF6BF1747F4AD2306D93 \
        FD3A5288F042B6850C66B31F09FE44734EB7990E \
        71DCFD284A79C3B38668286BC97EC7A07EDE3FC1 \
        DD8F2338BAE7501E3DD5AC78C273792F7D83545D \
        B9AE9905FFD7803F25714661B63B535A4C206CA9 \
        C4F0DFFF4E8C1A8236409D08E73BC641CC11F4C8 \
    ; do \
        gpg --keyserver ha.pool.sks-keyservers.net --recv-keys \
            "$key"; \
    done
ENV NODE_VERSION 4.4.5
```

```

RUN curl -SLO
"https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION
-linux-armv7l.tar.gz" \
&& curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/
SHASUMS256.txt.asc" \
&& gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256
.txt.asc \
&& grep "node-v$NODE_VERSION-linux-armv7l.tar.gz\$"
SHASUMS256.txt | sha256sum -c - \
&& tar -xzf "node-v$NODE_VERSION-linux-armv7l.tar.gz" -C /
usr/local --strip-components=1 \
&& rm "node-v$NODE_VERSION-linux-armv7l.tar.gz" SHASUMS256.
txt.asc SHASUMS256.txt

COPY scripts/rpi_docker/entrypoint /entrypoint
RUN mkdir /aaaida
WORKDIR /aaaida
CMD ["/entrypoint"]
EXPOSE 40000
COPY . /aaaida

```

Al principio del Dockerfile se ve la acción `FROM`, encargada de cargar una imagen de una Debian para ARM. Como observación se puede detectar que la imagen del mongo es para Ubuntu ARM y la imagen base para Aaaida es una Debian ARM, distribuciones de linux diferentes en los dos contenedores. Esto demuestra que cada contenedor es un ente aislado que no tiene que depender del resto de contenedores.

A continuación de declarar la imagen base, vienen una serie de acciones con la instrucción `RUN`, que nos permite ejecutar cualquier comando. En el primer caso que hace un update e instala el curl.

Las siguientes tres instrucciones de las cuales dos de ellas son un `RUN` y la restante un `ENV`, que configura las variables de entorno, son para poder instalar el Node.js en el contenedor para poder compilar el código de Aaaida.

Una vez instalado se utiliza la acción `COPY`, que como su nombre indica, copiará el contenido de un directorio en un directorio del contenedor.

Si seguimos se ve que crea un directorio con el nombre de `aaaida` y hace que ese directorio sea el directorio de trabajo con la instrucción `WORKDIR`.

Después Ejecutará `[/entrypoint]` que al escribirlo en formato JSON la instrucción `CMD` ejecuta el contenido sin shell. El contenido de este script especifica los puertos, el host... de la aplicación. Se podrá ver su contenido en el apéndice.

Para terminar con las 2 últimas instrucciones, que son `EXPOSE` y `COPY`, la cual la primera indica los puertos que el contenedor tendrá activos y por los cuales escuchará. El segundo hará una copia desde el punto raíz donde se ejecutará el Dockerfile en el directorio creado anteriormente de `aaaida`.

3.3.3. Despliegue de la aplicación

Para realizar el despliegue de Aaaida, con las dos imágenes que se han creado en el apartado anterior será suficiente. Tan solo tendremos que arrancar las imágenes para crear los contenedores. Primero se hará un `docker run` sobre la imagen de MongoDB y acto seguido igual con la imagen de Aaaida, pero con una pequeña diferencia: habrá que linkear el contenedor de mongo para que la aplicación pueda encontrar la base de datos necesaria para funcionar.

El comando utilizado para levantar el contenedor sería el siguiente:

```
docker run --link=cc1d12ed96ee:mongo --name= aaaidaArm -e NODE_ENV=docker
alteraid/aaaida datastore-arm
```

En el comando se linkea el `CONTAINER ID` que sería como el número de serie del contenedor de Docker (se puede obtener usando el comando `docker ps`), también le dará un nombre al contenedor y por último el `-e` que permite añadir variables de entorno.

Pero levantar los contenedores utilizando solo Docker implica estar haciendo un `run` o `start` cada vez que el servicio caiga por culpa de un apagón o fallos. Se decide instalar Docker Compose para poder orquestar las dos imágenes y añadir todas las necesidades y relaciones que necesiten. Esto permitirá tener el servicio automatizado en un fichero que tan solo habrá que arrancar una vez y en caso de fallo, el solo intentara arrancar de nuevo el servicio. El fichero utilizado se puede ver en el punto 2.5.3. Docker Compose donde también está explicado.

Si se levanta el proceso mediante el comando `docker-compose up` se puede ver como tanto el contenedor de `aaaida` como el de `mongo` se ejecutan y tenemos la aplicación corriendo perfectamente.

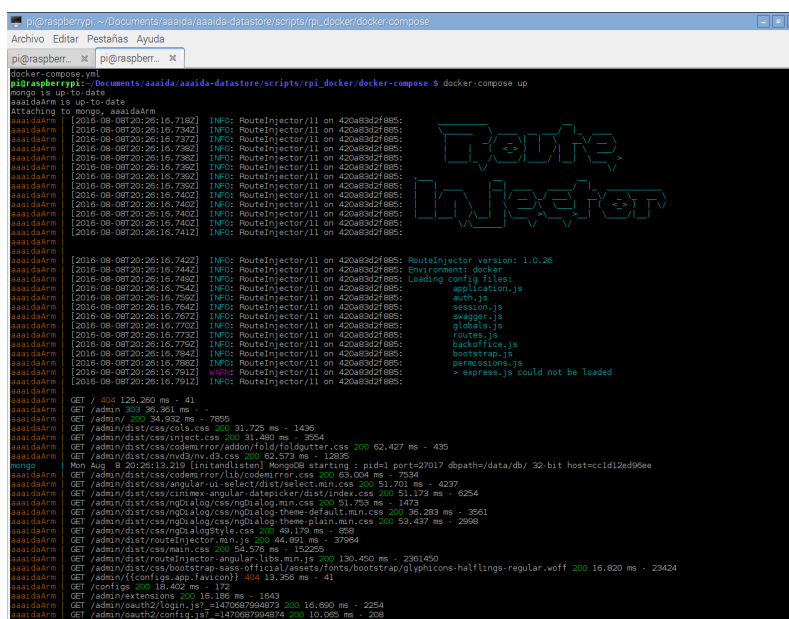
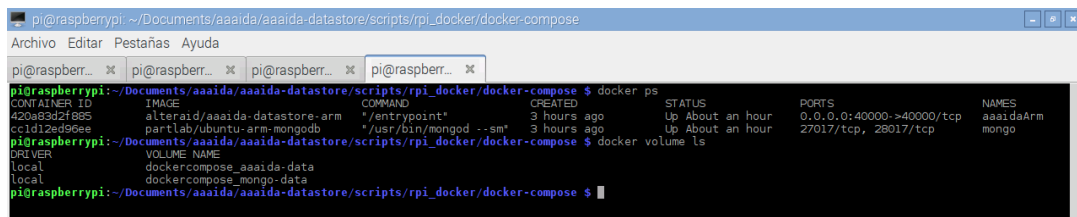


Figura 3.2: Docker-compose up



```

pi@raspberrypi: ~/Documents/aaaaida/aaaaida-datastore/scripts/rpi_docker/docker-compose
Archivo Editar Pestañas Ayuda
pi@raspberrypi: ~/Documents/aaaaida/aaaaida-datastore/scripts/rpi_docker/docker-compose $ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                                NAMES
420a83d2f8b5   alteraid/aaaaida-datastore-arm      "/entrypoint"           3 hours ago   Up About an hour   0.0.0.0:40000->40000/tcp   aaaidaArm
cc1d12ed96ee   partlab/ubuntu-arm-mongodb         "/usr/bin/mongod --sm"  3 hours ago   Up About an hour   27017/tcp, 28017/tcp      mongo
pi@raspberrypi: ~/Documents/aaaaida/aaaaida-datastore/scripts/rpi_docker/docker-compose $ docker volume ls
DRIVER        VOLUME NAME
local         dockercompose_aaaaida-data
local         dockercompose_mongo-data
pi@raspberrypi: ~/Documents/aaaaida/aaaaida-datastore/scripts/rpi_docker/docker-compose $

```

Figura 3.3: Listado de contenedores y volúmenes

En las dos figuras anteriores (3.2 y 3.3) se puede ver cómo al iniciar el Docker Compose se levantan todos los contenedores y se crean los volúmenes de datos necesarios para arrancar la Aaaida, la cual si abrimos un navegador podremos acceder. (Figura 3.4)

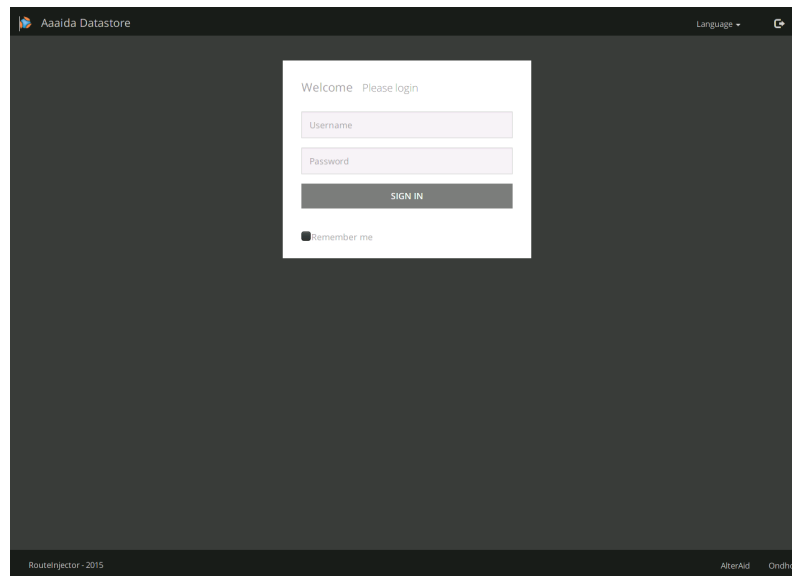
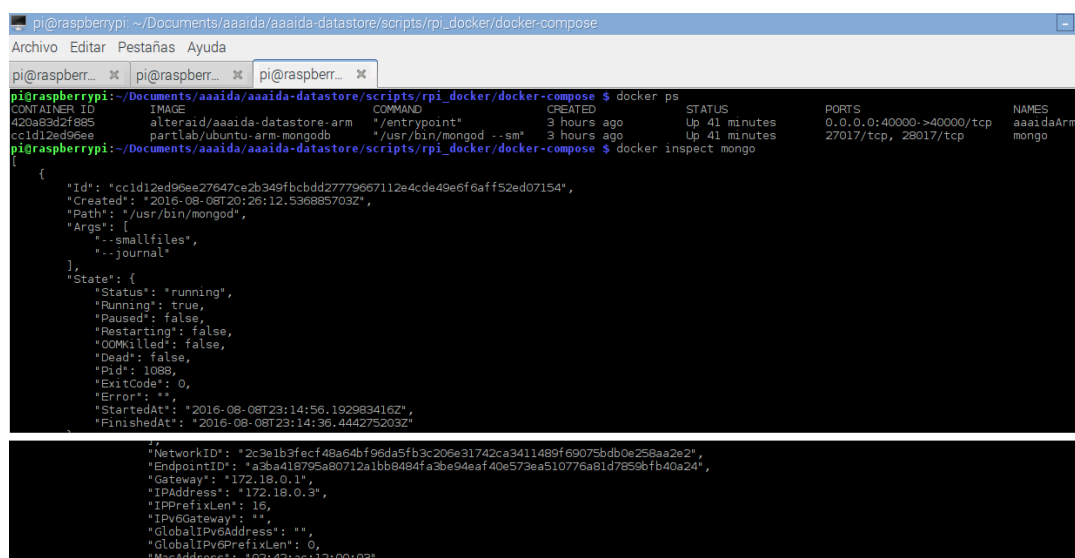


Figura 3.4: Acceso a Aaaida

La base de datos está vacía. Al no tener datos no hay usuarios por lo tanto no se podrá acceder a la plataforma de Aaaida, la solución es cargar un dump de la base de datos en el contenedor de `mongo` para que éste encuentre los datos y poder acceder a la plataforma. Para hacer el volcado de datos es necesario saber la IP del contenedor ya que será la manera de decir donde queremos hacer el restore de la base de datos. Para poder extraer información de los contenedores se utiliza el comando `docker inspect`. En la figura siguiente se verá la ejecución del comando y de donde obtendremos la IP de nuestro contenedor de mongo.



```

pi@raspberrypi: ~/Documents/aaaaida/aaaaida-datastore/scripts/rpi_docker/docker-compose
pi@raspberrypi:~/Documents/aaaaida/aaaaida-datastore/scripts/rpi_docker/docker-compose $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
420a83d2f885      alteraid/aaaaida-datastore-arm   "/entrypoint"       3 hours ago        Up 41 minutes      0.0.0.0:40000->40000/tcp   aaaaidaArm
cc1d12ed96ee      partlab/ubuntu-arm-mongodb      "/usr/bin/mongod -sm"  3 hours ago        Up 41 minutes      27017/tcp, 28017/tcp      mongo

pi@raspberrypi:~/Documents/aaaaida/aaaaida-datastore/scripts/rpi_docker/docker-compose $ docker inspect mongo
[
  {
    "Id": "cc1d12ed96ee27647ce2b349fbcbbdd27779667112e4cde49e6f6aff52ed07154",
    "Created": "2016-08-08T20:26:12.536885703Z",
    "Path": "/usr/bin/mongod",
    "Args": [
      "--smallfiles",
      "--journal"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 1088,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2016-08-08T23:14:56.192983416Z",
      "FinishedAt": "2016-08-08T23:14:36.444275203Z"
    },
    "NetworkID": "2c3e1b3fecf48a64bf96da5fb3c206e31742ca3411489f69075bdb0e258aa2e2",
    "EndpointID": "a3ba419795a80712a1bb484fa3be94eaf40e573ea510776a81d7859bfb40a24",
    "Gateway": "172.18.0.1",
    "IPAddress": "172.18.0.3",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:12:00:03"
  }
]

```

Figura 3.5: Docker inspect

Una vez se obtiene la IP del contenedor, `IPAddress: 172.18.0.3`, se ejecuta el siguiente comando para llevar a cabo el volcado de datos:

```
mongorestore dump/ -h 172.18.0.3
```

Una vez disponemos de todas las piezas listas, se puede dar por concluido el proceso de despliegue de Aaaida en una Raspberry Pi utilizando Docker.

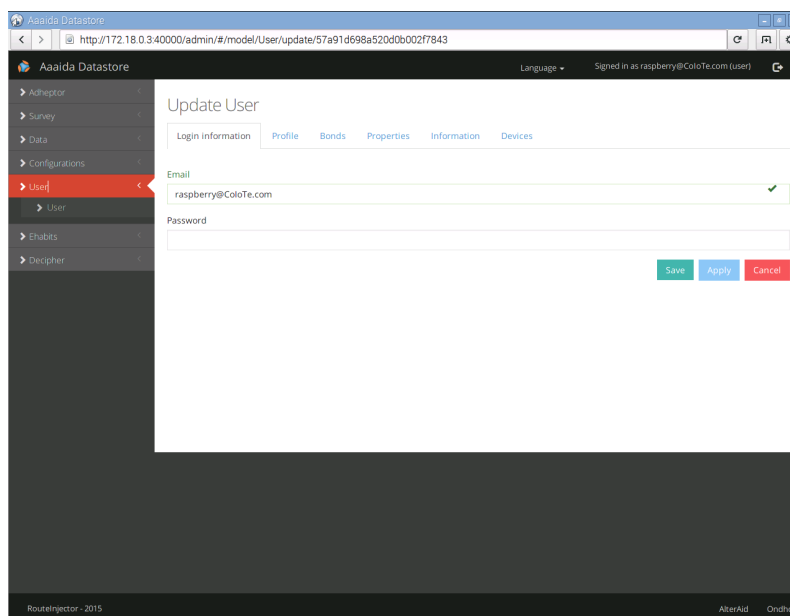


Figura 3.6: Interfaz de Aaaida

3.3.4. Conclusiones

En este capítulo se puede ver que la Raspberry Pi es el dispositivo para IoT más popular en el mercado, tanto por su prestaciones como por su reducido precio.

No es posible virtualizar la Raspberry Pi mediante Qemu y utilizar Docker ya que el kernel no es compatible.

Y por último, aunque Docker no sea compatible para un sistema ARM gracias a unos pequeños cambios en la imagen sí puede ejecutar. Por lo tanto se pudo desplegar toda la infraestructura en la Raspberry Pi y Aaaida funciona perfectamente.

CAPÍTULO 4. AAAIDA

En este capítulo se presentará el servicio de Aaaida utilizado para la creación de este proyecto.

4.1. Arquitectura

La arquitectura de Aaaida sería la que podemos ver en la figura 4.1 donde también podemos ver las tecnologías que utilizan cada una de las partes. Todas las partes que la componen utilizan JavaScript conocido como stack MEAN.

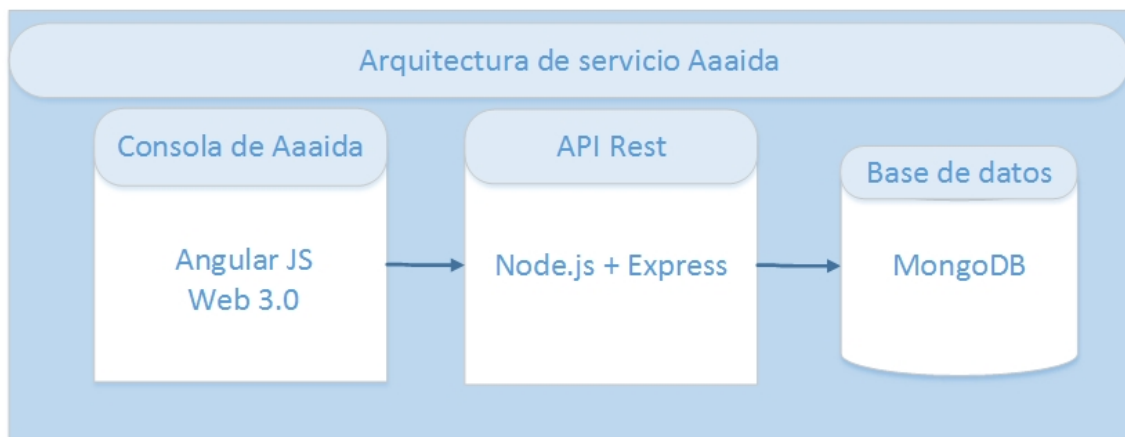


Figura 4.1: Arquitectura de Aaaida

4.1.1. Stack MEAN

El llamado stack MEAN es el desarrollo end-to-end basado en Javascript en cada una de sus partes, de esta manera, se permite desarrollar todo lo necesario sobre la infraestructura de JavaScript tal y como se puede ver en la figura 4.2

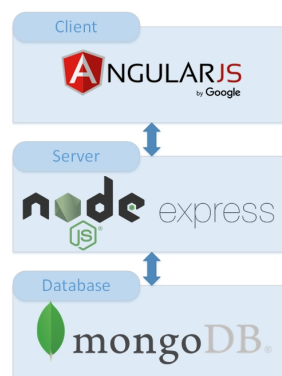


Figura 4.2: Stack MEAN

El hecho de que se utilice un mismo lenguaje de programación (JavaScript) para cada una de las tecnologías, permite que una persona pueda manejarse en cualquier ámbito, manteniendo una colaboración continua en los proyectos a desarrollar.

4.1.2. API REST

Una API Rest es una librería de funciones a la que se accede mediante el protocolo http, es decir, a través de direcciones web o URLs en las que se envían las consultas necesarias para acceder a la información que hay en la base de datos. Como respuesta a la consulta, se obtienen diferentes formatos que pueden ser textos planos, objetos JSON, entre otros.

Esta API está formada por los siguientes componentes y tecnologías:

4.1.2.1. MongoDB

Mongo es una base de datos no relacional (NoSQL) de código abierto que guarda cada uno de los datos en documentos JSON de forma binaria para que la integración sea más rápida. Orientado a documentos, de esquema libre, esto significa que cada entrada o registro puede tener un esquema de datos diferente, con atributos o “columnas” que no tienen por qué repetirse de un registro a otro.

4.1.2.2. Node.js

Node.js es un framework en JavaScript que utiliza el motor de Google denominado V8 y proporciona las funcionalidades core de una aplicación, mediante una arquitectura orientada a eventos asíncronos (APIs no-bloqueantes) que le permiten un gran rendimiento y escalabilidad. Aunque se puede utilizar para crear cualquier tipo de lógica de aplicación, el hecho de que disponga de un módulo para poder actuar como servidor web, hace que sea uno de los más utilizados en el desarrollo de aplicaciones web.

4.1.2.3. Express

Express es un framework en JavaScript para Node.js que permite crear servidores web y recibir peticiones http, de manera sencilla y eficiente. El objetivo principal de Express, es el de ofrecer soporte en diferentes necesidades, tales como: gestión de peticiones y respuestas, cabeceras...

4.1.3. Consola de Aaaida

La consola de Aaaida es una plataforma web desarrollada por Alteraid que permite administrar y controlar toda aquella información relacionada con las aplicaciones de la empresa.

Las tecnologías utilizadas las siguientes:

4.1.3.1. *AngularJS*

AngularJS es un framework mantenido por Google con JavaScript para la parte de cliente o frontend en una aplicación web. Este utiliza el patrón de diseño MVC (Modelo-Vista-Controlador) permitiendo crear SPAs (Single-Page Applications).

El hecho de que una aplicación web quepa en una sola página, proporcionando una experiencia fluida a cada uno de los usuarios. El patrón de arquitectura separe datos y lógica, permite el desarrollo este tipo de aplicaciones de una manera más flexible, convirtiéndose en una de las tecnologías más utilizadas.

4.1.3.2. *Web 3.0*

Web 3.0 es una web con la que interactuar para conseguir resultados, permitiendo compartir información por cada persona de forma inteligible y diseñada de manera eficiente con tiempos de respuesta optimizados. Este tipo de web facilita la accesibilidad de las personas a la información sin depender de qué tipo de dispositivo se use.

CAPÍTULO 5. IMPLEMENTACIÓN

En este capítulo se explicará las implementaciones realizadas para poder llevar a cabo el proyecto. Como conseguir la comunicación con el sensor, los cambios realizados en Aaaida, para poder visualizar las mediciones del sensor y por último el despliegue en la Raspberry Pi.

5.1. Comunicación con el sensor

Para la realización del proyecto es necesario un sensor que se pueda comunicar y enviar los datos mediante bluetooth. En la empresa se dispone de una serie de sensores médicos de los cuales se utilizará un monitor de ritmo cardiaco, Zephyr BioHarness 3.



Figura 5.1: Sesor Zephyr BioHarness 3

El fabricante ofrece gran cantidad de documentación y una aplicación de prueba para los desarrolladores que quieran realizar productos y utilicen sus sensores. Pero todo está orientado a aplicaciones Android, las cuales son las más populares para este tipo de sensores.

Por lo tanto, después de buscar y ponerse en contacto con la empresa, no hay ningún tipo protocolo para establecer la comunicación y recibir los datos en JavaScript. Por lo que se decidió realizar uno utilizando toda la documentación y ejemplos para otros lenguajes.

Para la realización del código de comunicación fueron necesarios 2 módulos de Node.js uno que nos calcula el CRC y otro que establece una conexión bluetooth.

Módulos utilizados:

- crc
- bluetooth-serial-port

5.1.1. Protocolo

El código realizado para poder establecer la comunicaciones fue el siguiente:

Se cargan los módulos externos y se declaran las variables, la dirección MAC del sensor se pone a clave para evitar interferencias con otros dispositivos bluetooth.

```
var crc = require('crc');
var btSerial = new (require('bluetooth-serial-port')).
  BluetoothSerialPort();

ADDRESS = "E0:D7:BA:A7:F1:5D";
var results = [];
var is_stopping = false;
```

Función connect, como su nombre indica, nos conectara con el sensor y empezará a recibir datos.

```
function connect(callback) {
  var socket = btSerial.on('found', function (address) {
    if (address == ADDRESS) {
      btSerial.findSerialPortChannel(address, function
      (channel) {
        btSerial.connect(address, channel, function
        () {
          console.log('connected to ' + address);
          btSerial.on('data', function (buffer) {
            decode(buffer);
          });
          listener(socket, function (res) {
            callback(res);
          });
        }, function () {
          console.log('cannot connect');
        });
      }, function () {
        console.log('found nothing');
      });
    }
  });
  btSerial.inquire();
}
```

La función decode, nos permite decodificar de una manera muy simplificada los bytes que se reciben. Le pasa el segundo byte y según su valor se puede saber qué tipo de mensaje envía el sensor. Si en el segundo byte que se recibe es un 44 implica que en el noveno byte que se está recibiendo el ritmo cardíaco.

```

function decode(data) {
  switch (data[1]) {
    case 35:
      console.log("Received LifeSign message");
      break;
    case 44:
      console.log("Received Event message");
      results.push(data[8]);
      break;
    case 43:
      console.log("Received Summary Data Packet");
      break;
    case 37:
      console.log("Received Accelerometer Data Packet"
);
      break;
    case 36:
      console.log("Received R to R Data Packet");
      break;
    case 33:
      console.log("Received Breathing Data Packet");
      break;
    default:
      console.log("Packet type: " + data[1]);
      console.log("Received Not recognised message");
      break;
  }
}

```

Una vez conectados al sensor se debe enviar mensajes a este para que mantenga la conexión y no se desconecte (lifeSings). Como la finalidad es realizar una medida la comunicación se realizará durante 20 seg, una vez pasados estos 20 seg se cerrara la conexión.

```

function listener(socket, callback) {
  socket.on('data', function (buffer) {
    decode(buffer);
    lifeSing = create_message_frame('100011', 0);
    socket.write(new Buffer(lifeSing), function (err,
bytesWritten) {
      if (err) console.log(err);
    });
  });
}

```

```

    setTimeout(function () {
        stop(socket, function (res) {
            callback(res);
        });
    }, 20000);
}

```

La creación de los mensajes lifeSings se realizan de la siguiente manera: Son la concatenación de un byte de sincronismo, el message id, el dlc que será la longitud del payload, el crc calculado mediante el payload y por ultimo otro byte de cierre. Todo el paquete se pasará a hexadecimal y se procederá a enviarlo.

```

function create_message_frame(message_id, payload) {
    dlc = payload.toString().length;
    if (0 <= dlc <= 128) {
        crc_byte = crc.crc32(payload);
        message_bytes = '00000010' + message_id + dlc +
        payload + crc_byte
        + '00000011';
        message_fame = Bin2Hex(message_bytes);
        return message_fame
    }
}

```

Por último la función de stop y la función de avg, la función de stop cerrará la conexión y la función de avg calcula la media de todas las medidas tomadas durante los 20 seg y devolverá la media.

```

function stop(socket, callback) {
    is_stopping = true;
    socket.close();
    avg(function (res) {
        callback(res);
    });
}

function avg(callback) {
    var sum = 0;
    if (is_stopping == true) {
        for (var i = 0; i < results.length; i++) {
            sum = sum + results[i];
        }
        var media = sum / results.length;
        callback(media);
    }
}

```


Con esto se puede establecer una conexión con el sensor y poder recibir la media del pulso cardiaco durante 20 seg. Hay que tener en cuenta la simplicidad del protocolo, ya que solo se captura una de las funciones (ritmo cardíaco) que proporciona el sensor Zephyr, ya que si se quiere obtener todos los datos la complejidad sería mucho mayor y para desarrollarlo en JavaScript se necesitaría mucha más información sobre cómo se envían las tramas y que contiene cada una de ellas.

5.2. Integración con Aaaida

Como se explicó en el capítulo anterior la consola de Aaaida es una plataforma web, que nos permite administrar toda la información de las aplicaciones vinculadas a ella. Para poder incluir nuestra aplicación de monitorización del ritmo cardiaco con Aaaida y así visualizar lo en la web hay que crear un plugin de Aaaida con nuestro proyecto.

5.2.1. Creación de un plugin

Para el desarrollo de cada uno de los plugins se ha utilizado el framework llamado route-injector desarrollado por Alteraid y Ondho.

El objetivo principal de este framework es el de generar, de manera automática: los método http CRUD de la API REST (GET, PUT, POST,DELETE) y una backoffice, mediante modelos.

Para la creación del plugin de proyecto deberemos de seguir la arquitectura principal de los plugins.

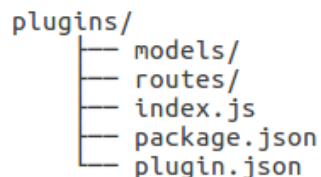


Figura 5.2: Arquitectura de los plugins

5.2.1.1. Models

En el caso de este proyecto no se creará ningún modelo de datos nuevo, ya que Aaaida dispone de un conjunto de modelos de datos que concuerda con el modelo que se necesita. El modelo elegido será Value, que cumple todas las necesidades de las medidas realizadas.

Los campos utilizados son los siguientes:

```
{
  value: {type: String},
  user: {type: mongoose.Schema.Types.Mixed, ref: 'User'},
  bond: {type: mongoose.Schema.Types.Mixed, ref: 'Bond'},
  measure: {type: mongoose.Schema.Types.Mixed, ref: 'Measure'},
  tags: {type: String},
  measured_at: {type: Date, readonly: true}
}
```

- value: Será el valor de la medida de ritmo cardiaco registrada.
- user: El usuario con el cual se está logueado en Aaaida
- bond: Sería el “paciente” al cual se le toma la medida.
- tags: Este campo se utiliza para diferenciar de qué aplicación pertenecen los valores.
- measured at: El instante que se tomó el valor.

5.2.1.2. Routes

En el caso de que fuese necesario obtener información más concreta sobre el modelo de datos, se deberían de definir cada una de esas rutas en la carpeta routes del plugin. Como se dijo, route-injector genera automáticamente el CRUD para el modelo Values, pero necesita una ruta específica para ejecutar la conexión vía bluetooth con el sensor.

Por lo tanto fue necesario la creación de la ruta, en el mismo fichero se copio todo el protocolo para la conexión con el sensor Zephyr.

```
module.exports.route = function (router) {
  router.get('/coiote/media', function (req, res) {
    connect(function (media) {
      console.log("Tu HR media = " + media);
      res.json(media)
    });
  });
};
```

Como se puede apreciar en el código una petición `get a /coiote/media` ejecutará la función `connect` que establece la comunicación y recolección de datos.

5.2.1.3. Index

En el siguiente fichero se configuran las funciones iniciales del plugin una vez este ha cargado.

```
module.exports.config = require('./plugin.json');  
  
module.exports.init = function (conf) {  
};
```

El plugin no debe hacer ninguna función al iniciarse, por lo tanto, no debemos añadir nada en este fichero.

5.2.1.4. Package

En este fichero, se describe toda la información necesaria del paquete.

```
{  
  "name": "coiote-plugin",  
  "version": "0.0.1",  
  "description": "Plugin for CoIoTe",  
  "dependencies": {  
    "bluetooth-serial-port": "^2.0.0",  
    "crc": "^3.4.0"  
  }  
}
```

Se puede apreciar las 2 dependencias a módulos externos nombrados anteriormente.

5.2.1.5. Plugin

En este fichero, se indica el nombre del plugin y el directorio donde se encuentran las rutas estáticas. Sirve para especificar dónde se encuentran todas esas rutas generadas para el plugin y no han sido creadas automáticamente por route-injector.

```
{  
  "name": "CoIoTe",  
  "routes": ["routes"]  
}
```

5.2.2. Creación de la página

Una vez el plugin está creada, se necesitará crear la página web donde poder visualizar en la consola de Aaaida los resultados obtenidos de las medidas.

Para esto se seguirá un proceso similar al de crear un plugin, pero en el directorio de pages, que tiene una arquitectura base similar a la siguiente:

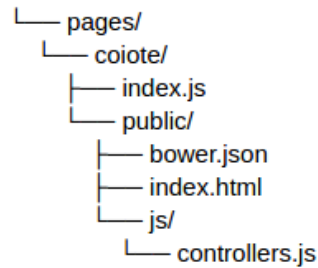


Figura 5.3: Arquitectura del directorio pages

En este directorio, se encuentran definidos cada uno de los templates externos que se añadirán a la backoffice.

En este caso, ha sido necesario añadir el de coiole. En este directorio podemos encontrar el template index.js donde se define el controlador y como se visualiza en la consola de Aaaida y su URL.

```

module.exports = {
  backoffice: true,
  url: 'coiole/mesures',
  template: 'coiole/index.html',
  controller: 'ChartCoioleController',
  menu: {
    clickTo: 'coiole/mesures',
    title: "mesures",
    section: "CoIoTe"
  }
}
//backoffice: false // standalone website
};

```

Creandonos una pestaña en la consola de Aaaida para el proyecto como podemos ver en la figura 5.4.

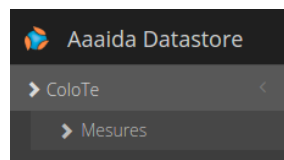


Figura 5.4: Sección en la consola de Aaaida

Una vez creado este fichero pasaremos a crear la template index.html y su controlador controller.js. En el controlador se realizan petición a las rutas creadas por route-injector para el modelo value y la ruta creada en el directorios routes para establecer la comunicación, y así poder rellenar el contenido de la plantilla.

La página para la visualización de los resultados sería esta:

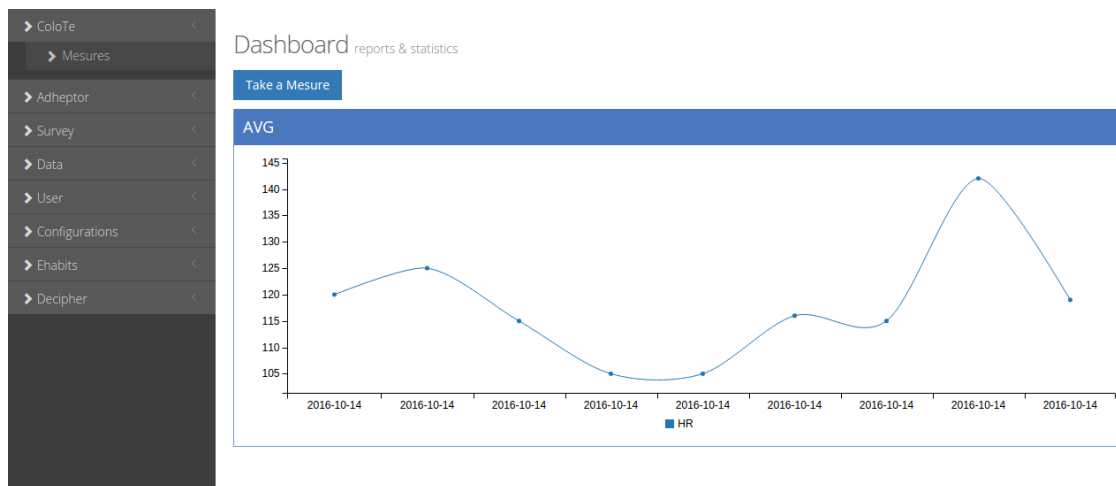


Figura 5.5: Visualización de los datos en Aaaida

5.2.3. Recursos utilizados

A continuación se mostrará en una tabla todos aquellos recursos de la API de Aaaida que fueron utilizados.

Modelo	Tipo	Ruta
Bond	get	/bond/:user.login
Value	post	/values
Value	post	/value
-	get	/coiote/media

Cuadro 5.1: Tabla de rutas

- Devuelve toda la información del usuario que está logueado en Aaaida.
- Retorna los valores del bond del cual hemos tomado la medida.
- Guarda el valor de la medida realizada.
- Establece la conexión y la recogida de datos.

5.3. Despliegue en la Raspberry Pi

Para realizar el despliegue de la aplicación, se llevaron a cabo un seguido de cambios. Lo primero a realizar, fue la creación de un módulo npm con el plugin de coiote. Para poder resolver las dependencias necesarias al cargar Aaaida. Resuelto el problema con las dependencias, el siguiente cambio realizado fue en el Dockerfile, para crear la imagen de

Aaaida. Se añadió la instalación de las librerías para el uso del bluetooth en la Raspberry Pi, necesarias para la comunicación con el sensor.

Las librerías instaladas fueron:

- libbluetooth3
- bluez

El siguiente cambio que se llevó a cabo fue en el `docker-compose.yml`. Para que el contenedor de Docker pueda utilizar todas las redes visibles por la máquina donde esté instalado, se debe de añadir dos argumentos.

- `privileged: true`: Permite desactivar el etiquetado de seguridad, dando acceso total a la máquina.
- `network mode: "host"`: Permite ver todas las redes disponibles por la máquina.

El problema, al añadir el argumento `network mode` este contenedor no podrá utilizar el argumento `link`, que vincula el contenedor de `aaaida` con el contenedor de `mongo`. Ya que Docker no soporta las dos funcionalidades activas, se procedió a eliminar ese `link` que unía los dos contenedores. Para solucionar el problema, ya que es imprescindible que el contenedor de `aaaida` pueda acceder a los datos se tiene que exponer los puertos del servicio de `mongo` para que este sea visible desde el exterior. A si el contenedor de `aaaida` no sabrá de la existencia de otro contenedor con sus datos pero como los puertos de este contenedor están abiertos se pueden realizar las peticiones. Estos cambios implican que se deben de hacer modificaciones en los ficheros de `env`, para que `aaaida` no haga las peticiones al contenedor linkado sino a la IP y al puerto abierto.

Los ficheros modificados se podrán ver en los Apéndices del proyecto.

Una vez resuelto todos los problemas de comunicación, se procederá a el despliegue de de toda la infraestructura y puesta en marcha:

- Desde el directorio de `aaaida-datastore` se ejecutará un script el cual nos instalará todas las dependencias del proyecto y ejecutará el `docker build` de la imagen de Aaaida. (Este script se podrá ver en apéndices)

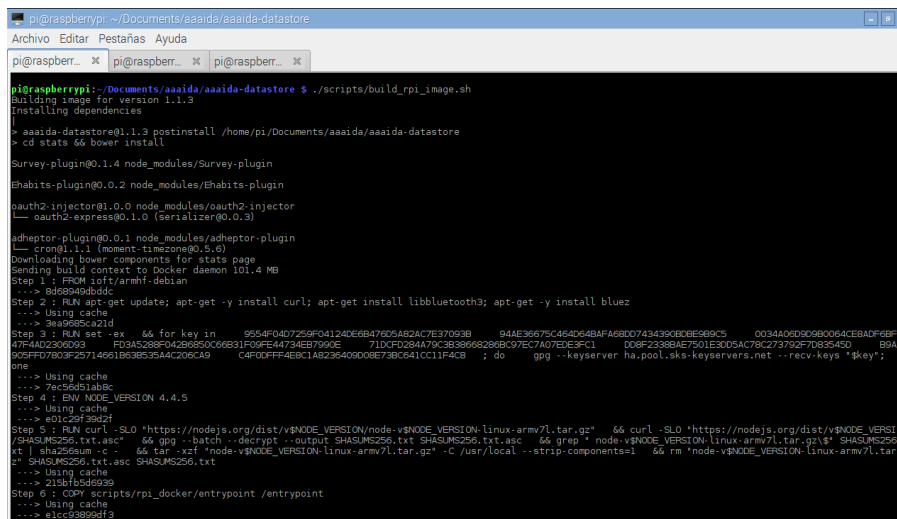


Figura 5.6: Build del script

- Una vez generada la imagen, procederemos a ejecutar el `docker-compose.yml` desde su directorio. Generando los contenedores de `aaaيدا` y `mongo` con de sus volúmenes y conexiones.

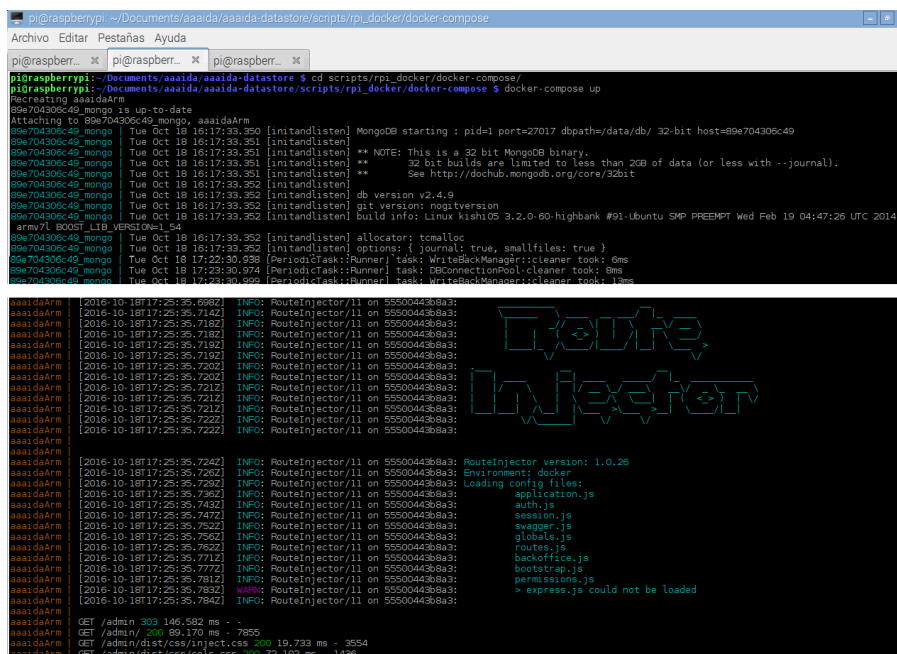


Figura 5.7: Creación de contenedores y ejecución

- Desplegado todos los contenedores y Aaaida funcionado, podemos acceder a la consola de Aaaida desde otro ordenador sabiendo la IP.

- Para realizar una medida, se debe clicar en el botón “Take a measure” este realizará la conexión con el Zephyr, que cuando empiece a medir se encenderá el led azul y estará realizando capturas durante 20 seg. Una vez la medida esté realizada, se nos mostrará la medida y otro botón para validarla, en caso que la medida sea correcta, guardandola en la base de datos.

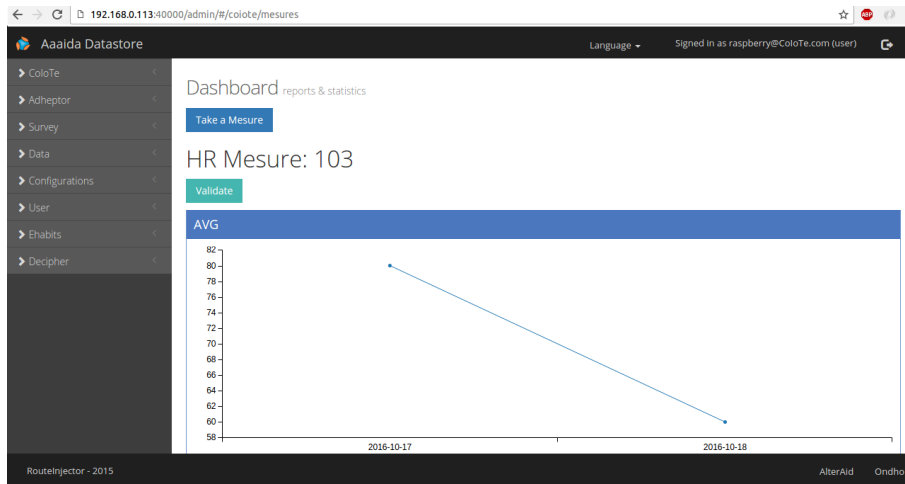


Figura 5.8: Tomar la medida

```
POST /values/ 200 478.264 ms - 1419
[2016-10-18T17:05:21.294Z] INFO: RouteInjector/20 on raspberrypi: connected to E0:D7:BA:A7:F1:5D
[2016-10-18T17:05:41.302Z] INFO: RouteInjector/20 on raspberrypi: Tu HR media = 103
GET /colote/media 200 34446.334 ms - 5
```

Figura 5.9: Conexión con el sensor

```
[2016-10-18T17:08:10.282Z] INFO: RouteInjector/20 on raspberrypi:
pre saving... { value: '103',
  user: { _id: 57a91d698a520d0b002f7843, email: 'raspberrypi@ColoTe.com' },
  tags: 'ColoTe',
  bond: { _id: 57a91d698a520d0b002f7845, name: 'Raspberry' },
  measure:
    { _id: 57ff9c7264aaac692db3f222,
      type: 'STRING',
      units: '',
      i18n: [ [Object] ] },
    _id: 580656fa03a9cc1400695aed,
    information: { updated_at: Tue Oct 18 2016 17:04:38 GMT+0000 (UTC) },
    comments: [],
    properties: [] }
POST /value 201 406.459 ms - 34
[2016-10-18T17:08:10.383Z] DEBUG: RouteInjector/20 on raspberrypi: PostDB finished
```

Figura 5.10: Guardar en la base de datos

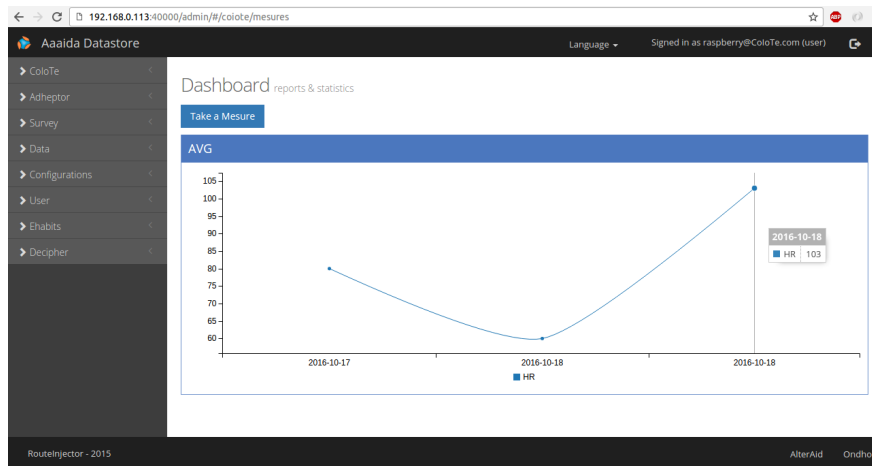


Figura 5.11: Visualización del resultado

En caso que no se disponga de conexión de internet, se debería modificar los ficheros de `/etc/network/interfaces` tanto de la Raspberry Pi como del PC que se utilizará para conectarse a ella. Añadiendo IPs estáticas y creando una red Ad-hoc.

5.4. Conclusiones y Resultados

Conseguir establecer la comunicación con el sensor fue una tarea para nada sencilla. Incluso haciendo tan solo la recogida de datos de una funcionalidad, se tuvo que entender muy bien cómo funcionaba y enviaba el sensor los datos sin contar que la documentación presentada por el fabricante se basaba únicamente en aplicaciones Android.

Como resultados, se desarrolló un protocolo de comunicación con el sensor, el cual nos proporcionaba la media del ritmo cardíaco. La comunicación se podía ejecutar mediante la consola de Aaaida donde se visualizará el resultado. Una vez validado se guardará en la base de datos y, para finalizar, un gráfico donde se puede ver un seguimiento de las medidas realizadas.

CONCLUSIONES

En este capítulo se comentarán las conclusiones finales y personales del proyecto realizado, así como los objetivos conseguidos y implementaciones futuras.

5.5. Conclusiones del proyecto

El propósito de este proyecto ha consistido en el desarrollo de un producto nuevo para la empresa AlterAid. Este producto realiza un despliegue del servicio Aaaida de la empresa en un dispositivo móvil, como sería una Raspberry Pi, que nos permitirá poder desplegarlo en lugares con difícil acceso o que hayan sufrido catástrofes y establecer una pequeña red de sensores. Para llevar a cabo el despliegue se utilizó Docker, para poder crear contenedores que contengan el software y nos faciliten la gestión.

Para el desarrollo del proyecto ha sido necesario definir una serie de objetivos, para lograr el avance. El primer objetivo consistió en analizar las diferentes tecnologías empleadas en el trabajo. Todo este análisis se ve reflejado satisfactoriamente en los primeros capítulos. Una vez el aprendizaje estaba concluido, se pasó a la instalación del entorno, donde se debía de instalar Docker en la Raspberry Pi, que como se puede ver en el capítulo 3 se consiguió realizar, se descubrió que no es posible virtualizar la Raspberry Pi con Qemu y utilizar Docker. Era necesario disponer de una Raspberry Pi 3 para la realización del proyecto. Con todo el entorno funcionando perfectamente, se realizó el despliegue de Aaaida en el dispositivo. Donde también se obtuvieron resultados positivos. Para la creación de los nodos, se utilizó un sensor de monitorización del ritmo cardíaco, para la toma de medidas y la comunicaciones con la Raspberry Pi mediante bluetooth. Este fue un punto crítico del proyecto, ya que aun disponiendo de documentación y alguna aplicación de ejemplo, se tuvo que crear un protocolo para establecer la comunicación y captar los datos del sensor. Cosa que llevo mucho tiempo para lograr entender cómo se transmitían los datos. Para finalizar, se realizó un plugin para Aaaida y así poder captar, visualizar y almacenar los datos tomados por el sensor.

El proyecto concluye de forma satisfactoria ya que se ha conseguido cumplir todos los puntos establecidos de manera aceptable.

La utilización de Docker para un despliegue, fuera del mundo de los servidores, queda probado que es posible. Es Viable siempre que se puedan salvar los inconvenientes mostrados a lo largo de proyecto. Las mejoras obtenidas gracias a la utilización de Docker son: su facilidad y comodidad a la hora de realizar despliegues de las aplicaciones y su mantenimiento ya que la actualización de estas se puede hacer parcialmente gracias a sus separación por contenedores.

5.6. Conclusiones personales

Gracias al desarrollo de este trabajo, he adquirido tanto conocimientos sobre nuevas tecnologías como también, me ha permitido mejorar la manera de organizarme a la hora de desarrollar proyectos.

Los conocimientos conseguidos con la realización del proyecto tanto sobre las tecnologías empleadas como en la manera de distribuir tareas son de gran ayuda de cara a mi vida laboral y personal.

La realización del proyecto me ha resultado muy interesante, ya que he podido realizar un trabajo utilizando tecnologías nuevas para mí. Cosa que me puso a prueba, a la hora de tener la necesidad de adquirir conocimiento de manera autónoma.

5.7. Implementaciones futuras

Se puede afirmar que los objetivos principales de este proyecto se han cumplido. De todas maneras, existen mejoras que pueden dar un valor añadido, tales como:

- Incluir nuevos sensores para la toma de medidas.
- Creación de nuevas aplicaciones o maneras de tomar las medidas.

Como implementación futura de carácter personal, se podría mejorar el protocolo de conexión con Zephyr y publicarlo como módulo npm para Node.js ya que no existe ningún tipo de conector en JavaScript.

5.8. Impacto ambiental

El análisis sobre el impacto ambiental del proyecto. Dado que se centra en desarrollo de software lo único a tener en cuenta sería la batería del sensor y que la Raspberry Pi necesita una toma de corriente para poder funcionar. Donde hay que añadir que los dos dispositivos utilizados, son de bajo consumo.

GLOSARIO

IoT: Internet Of Things

Journalig: Memoria caché en la cual se tiene constancia de las escrituras hechas en disco. Esto en caso de fallo o desconexión no prevista ayuda a evitar la corrupción de datos. Por esta razón se activa, ya que la Raspberry Pi es un dispositivo sin batería y si hay un corte de luz, puede que se corrompa la base de datos y el restar no se pueda ejecutar.

Daemon: Proceso en segundo plano que se inicia como servicio.

API: Application Programming Interface. Es un conjunto de especificaciones para la comunicación entre diferentes componentes software.

CRC: Código de detección de errores.

Payload: Los datos transmitidos en una comunicación.

JSON: Javascript Object Notation.

MVC: El modelo vista controlador, es un patrón de arquitectura de software que separa los datos y la lógica de una interfaz de usuario y el módulo encargado de gestionar cada uno de sus eventos y comunicaciones.

SPA: Single-page Application, es una aplicación web que cabe en una sola página con el objetivo de proporcionar una experiencia fluida a los usuarios.

BIBLIOGRAFÍA

- [1] Turnbull, J. *The Docker book*. (último acceso: 20 de 06 de 2016)
- [2] Humble, J. *Continuous Delivery*. (último acceso: 20 de 06 de 2016)
- [3] Newman, S. *Building Microservices*. (último acceso: 20 de 06 de 2016)
- [4] **Web de Raspberry Pi**. URL: <https://www.raspberrypi.org/> (último acceso: 17 de 10 de 2016)
- [5] **Web de Docker**. URL: <https://www.docker.com> (último acceso: 17 de 10 de 2016)
- [6] **Blog de Hypriot**. URL: <http://blog.hypriot.com/> (último acceso: 17 de 10 de 2016)
- [7] **Web de Alteraid**. URL: <http://www.alteraid.com/> (último acceso: 17 de 10 de 2016)
- [8] **Web de MongoDB**. URL: <https://www.mongodb.com/> (último acceso: 17 de 10 de 2016)
- [9] **Repositorio de Angular-chart**. URL: <http://jtblin.github.io/angular-chart.js/> (último acceso: 17 de 10 de 2016)
- [10] **Repositorio de CRC**. URL: <https://www.npmjs.com/package/crc> (último acceso: 17 de 10 de 2016)
- [11] **Repositorio de bluetooth-serial-port** . URL: <https://www.npmjs.com/package/bluetooth-serial-port> (último acceso: 17 de 10 de 2016)
- [12] **Web de Zephyr, development tools** . URL: <https://www.zephyranywhere.com/zephyr-labs/development-tools> (último acceso: 17 de 10 de 2016)
- [13] **Web de Qemu** . URL: http://wiki.qemu.org/Main_Page (último acceso: 17 de 10 de 2016)
- [14] **Web de Vmware** . URL: <http://www.vmware.com> (último acceso: 17 de 10 de 2016)
- [15] **GitHub** . URL: <https://github.com> (último acceso: 17 de 10 de 2016)

APÉNDICES

APÉNDICE A. EMULAR LA RASPBERRY PI EN QEMU

```
----- Compilar qemu con soporte ARM 1176 ----

# apt-get install git zlib1g-dev libssl1.2-dev
# apt-get install libpixman-1-dev libfdt-dev
$ mkdir raspberrypi
$ cd raspberrypi
$ git clone git://git.qemu-project.org/qemu.git
$ cd qemu
$ ./configure --target-list="arm-softmmu arm-linux-user"
--enable-sdl --prefix=/usr
$ make
# make install

---- Descargar la imagen de la raspberry pi y el kernel-
qemu -----

$ wget "http://downloads.raspberrypi.org/raspbian_latest"
$ unzip 2016-03-18-raspbian-jessie.img.zip
$ rm -rf 2016-03-18-raspbian-jessie.img.zip

$ fdisk -l 2016-03-18-raspbian-jessie.img
$ sudo mount -v -o offset=67108864 -t ext4 2016-03-18-
  raspbian-jessie.img /mnt/
$ cd /mnt
$ sudo nano ./etc/ld.so.preload //(comentar el codigo)
$ sudo nano ./etc/fstab //(Comentar la segunda linea)
$ cd ~
$ sudo umount /mnt

----- Ejecutar por primera vez -----

$ qemu-system-arm -kernel kernel-qemu -cpu arm1176 -m 256 -
  M versatilepb -no-reboot -serial stdio -append "root=/
  dev/sda2 panic=1 rootfstype=ext4 rw" -hda 2016-03-18-
  raspbian-jessie.img -redir tcp:2222::22
```

```
----- Resize de la imagen -----  
  
$ qemu-img resize raspi.img +16024M  
$ qemu-system-arm -M versatilepb -cpu arm1176 -hda  
  2016-03-18-raspbian-jessie.img -kernel kernel-qemu -m  
  192 -append "root=/dev/sda2" -drive file=raspi.img  
$ sudo apt-get -y install gparted  
$ sudo gparted  
$ qemu-system-arm -kernel kernel-qemu -cpu arm1176 -m 256 -  
  M versatilepb -no-reboot -serial stdio -append "root=/  
  dev/sda2 panic=1 rootfstype=ext4 rw" -hda raspi.img -  
  redirect tcp:2222::22
```

Puede haber problemas con la interfaz gráfica, ya que si no hay suficiente memoria en la imagen, se deshabilita.

APÉNDICE B. FICHEROS UTILIZADOS

Todos los ficheros utilizados y modificados para realizar el despliegue.

B.1. Fichero entrypoint

```
function wait {
    echo -n "waiting for TCP connection to $1:$2..."
    while ! nc -w 1 $1 $2 2>/dev/null
    do
        echo -n .
        sleep 1
    done
    echo 'ok'
}

function startup {
    list=$(env | grep DOCKER_ | grep _TCP= | cut -d = -f 2)
    elems=( $list )
    for key in "${!elems[@]}"
    do
        str=${elems[$key]}
        str2=${str#"tcp://"}
        IFS=:
        array=( $str2 )
        unset IFS
        host=${array[0]}
        port=${array[1]}
        wait $host $port
    done
}

startup
node bin/www
```

B.2. Script build rpi image

```
#!/bin/bash
getVersion() {
    echo $(node -p -e "require('./package.json').version");
}
echo "Building image for version $(getVersion)"
echo "Installing dependencies"
npm install
echo "Downloading bower components for stats page"
```

```
cd stats
bower install --allow-root
cd ..
VERSION=getVersion
docker build -t alteraid/aaaaida-datastore-arm -f scripts/
  rpi_docker/Dockerfile .
```

B.3. Dockerfile modificado

```
FROM ioft/armhf-debian
RUN apt-get update; apt-get -y install curl;
apt-get install libbluetooth3; apt-get -y install bluez

RUN set -ex \
    && for key in \
        9554F04D7259F04124DE6B476D5A82AC7E37093B \
        94AE36675C464D64BAFA68DD7434390BDBE9B9C5 \
        0034A06D9D9B0064CE8ADF6BF1747F4AD2306D93 \
        FD3A5288F042B6850C66B31F09FE44734EB7990E \
        71DCFD284A79C3B38668286BC97EC7A07EDE3FC1 \
        DD8F2338BAE7501E3DD5AC78C273792F7D83545D \
        B9AE9905FFD7803F25714661B63B535A4C206CA9 \
        C4F0DFFF4E8C1A8236409D08E73BC641CC11F4C8 \
    ; do \
        gpg --keyserver ha.pool.sks-keyservers.net --recv-keys
        "$key"; \
    done

ENV NODE_VERSION 4.4.5

RUN curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/node-
  v$NODE_VERSION -linux-armv7l.tar.gz" \
    && curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/
  SHASUMS256.txt.asc" \
    && gpg --batch --decrypt --output SHASUMS256.txt
  SHASUMS256.txt.asc \
    && grep " node-v$NODE_VERSION-linux-armv7l.tar.gz\$"
  SHASUMS256.txt | sha256sum -c - \
    && tar -xzf "node-v$NODE_VERSION-linux-armv7l.tar.gz" -C
  /usr/local --strip-components=1 \
    && rm "node-v$NODE_VERSION-linux-armv7l.tar.gz"
  SHASUMS256.txt.asc SHASUMS256.txt

COPY scripts/rpi_docker/entrypoint /entrypoint
RUN mkdir /aaaaida
```

```
WORKDIR /aaaaida
CMD ["/entrypoint"]

EXPOSE 40000

COPY . /aaaaida
```

B.4. Docker-compose.yml modificado

```
version: '2'
services:
  mongo:
    container_name: mongo
    restart: always
    image: partlab/ubuntu-arm-mongodb
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db
    command: /usr/bin/mongod --smallfiles --journal
  aaaaida:
    container_name: aaaaidaArm
    restart: always
    network_mode: "home"
    privileged: true
    image: alteraid/aaaaida-datastore-arm
    ports:
      - "40000:40000"
    environment:
      - NODE_ENV=docker
    volumes:
      - aaaaida-data:/mnt/aaaida.js/

volumes:
  mongo-data:
    driver: local
  aaaaida-data:
    driver: local
```